NEURAL NETWORKS AS ORDINARY DIFFERENTIAL EQUATIONS

Dec 11, 2018 · 7 minutes read

Recently I found a paper being presented at NeurIPS this year, entitled Neural Ordinary Differential Equations, written by Ricky Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud from the University of Toronto. The core idea is that certain types of neural networks are analogous to a discretized differential equation, so maybe using off-the-shelf differential equation solvers will help get better results. This led me down a bit of a rabbit hole of papers that I found very interesting, so I thought I would share a short summary/view-from-30,000 feet on this idea.

Typically, we think about neural networks as a series of discrete layers, each one taking in a previous state vector $\mathbf{h}_n$ and producing a new state vector $\mathbf{h}_{n+1} = F(\mathbf{h}_n)$. Here, let's assume that each layer is the same width (e.g. $\mathbf{h}_n$ and $\mathbf{h}_{n+1}$ have the same dimension, for every n). Note that we don't particularly care about what F looks like, but typically it's something like $F(x) = \sigma(\sum_i \theta_i x_i)$, where σ is an activation function (e.g. relu or a sigmoid), and θ is a vector of parameters we're learning. This core formulation has some problems - notably, adding more layers, while theoretically increasing the ability of the network to learn, can actually *decrease* the accuracy of it, both in training and test results.

This problem was adressed by Deep Residual Learning, from Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun out of Microsoft Reasearch. The idea in a nutshell is to learn a function of the difference between layers: $\mathbf{h}_{n+1} = F(\mathbf{h}_n) + \mathbf{h}_n$. In the paper, they show this simple transformation in what you're learning allows the networks to keep improving as they add more layers. To me, this reminds me of delta encoding, in which you represent a stream of data as a series of changes from the previous state.] !
This can make certain types of data much more suitable to compression (see, e.g. this article from Glenn Fiedler on compressing physics data to send over a network). It makes some kind of sense that if delta encoding can make data easier to compress, it could also make it easier to represent for a neural network.

Euler's method and residual networks

But how do residual networks relate to differential equations? Suppose we have some constant that we'll call $\Delta t \in \mathbb{R}$. Then we can write the state update of our neural network as

$$\mathbf{h}_{t+1} = F(\mathbf{h}_t) + \mathbf{h}_t$$

$$= \frac{\Delta t}{\Delta t} F(\mathbf{h}_t) + \mathbf{h}_t$$

$$= \Delta t G(\mathbf{h}_t) + \mathbf{h}_t$$

where now we're learning $G(\mathbf{h}_t) = F(\mathbf{h}_t)/\Delta t + \mathbf{h}_t$. If you have experience with differential equations, this formulation looks very familiar - it is a single step of Euler's method for solving ordinary differential equations. It seems this was first noticed by Weinan E in A proposal on Machine Learning via Dynamical Systems, and expanded upon by Yiping Lu et al. in Beyond Finite Layer Neural Networks. However, Lu et al. continue to treat the network as a series of discrete steps, and use a discrete solver with fixed timesteps to come up with a novel neural network architecture. The reason for this is that we need to be able to train the networks, and it's not really clear how to "learn" a differential system. Chen, Rubanova, Bettencourt and Duvenaud solve this problem by using some clever math which enables them to compute the gradients they need for backpropagation.

Evaluating ODEs

Before we get to that, let's look at what we're trying to solve. If we consider a layer of our neural network to be doing a step of Euler's method, then we can model our system by the differential equation

$$\frac{d\mathbf{h}(t)}{dt} = G(\mathbf{h}(t), t, \theta)$$

from before:
$$G(\vec{h}_t) = \frac{\vec{h}_{t+1} - \vec{h}_t}{\Delta t}$$

Here we've made explicit G's dependency on t, as well as some parameters θ which we will train on. In this formulation, the output of our "network" is the state $\mathbf{h}(t_1)$ at some time t_1. Therefore, if we know how to describe the function G, we can use any number of off-the-shelf ODE solvers to evaluate the neural network.

$$\mathbf{h}(t_1) = \text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta)$$

There is a ton of research on different methods that can be used as our ODESolve function, but for now we'll treat it as a black box. What matters is that if you substitute in Euler's method, you get exactly the residual state update from above, with

$$\text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta) = \mathbf{h}(t_0) + (t_1 - t_0)G(\mathbf{h}(t_0), t_0, \theta)$$

However, we don't need to limit ourselves to Euler's method, and in fact will do much better if we use more modern approaches.

Training the beast

So how to train it? Suppose we have a loss function

$$L(h(t_1)) = L(\text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta))$$

To optimize L, we require gradients with respect to its parameters $\mathbf{h}(t)$ (the state of our system at time t), t (our "time" variable, which is sort of a continuous analog to depth), and θ, our training parameters. The adjoint method describes a way to come up with this. The adjoint method is a neat trick which uses a simple substitution of variables to make solving certain linear systems easier. Part of the reason this paper grabbed my eye is because I've seen the adjoint method before, in a completely unrelated area: fluid simulation! In this paper from McNamara et al., they make controlling a fluid simulation easier by using the adjoint method to efficiently compute some gradients with respect to user controlled parameters. That certainly sounds similar to our problem.

$\rightarrow$ So what is the adjoint method? Suppose we have 2 known matrices A and C, and an unknown vector $\mathbf{u}$ and we would like to compute a product

$$\mathbf{u}^\intercal B \text{ such that } AB = C$$

We could first solve the linear system to find the unknown matrix B, then compute the product, but solving the linear system could be expensive. Instead, let's solve a different problem. Let's find a vector $\mathbf{v}$ and compute

$$\mathbf{v}^\intercal C \text{ such that } A^\intercal \mathbf{v} = \mathbf{u}$$

We can show that these are in fact the same problem:

$$\mathbf{v}^\intercal C = \mathbf{v}^\intercal AB = (A^\intercal \mathbf{v})^\intercal B = \mathbf{u}^\intercal B$$

Through this transformation, we've reduced the problem from solving for a matrix, and reduced it to solving for a vector. This can be a big computational win! So how do we use it to train networks? I'm not going to go into the complete details here as it's slightly involved, but Appendix B in the paper has the complete derivation. In brief, we define the adjoint state as

$$a(t) = -\partial L / \partial \mathbf{h}(t)$$

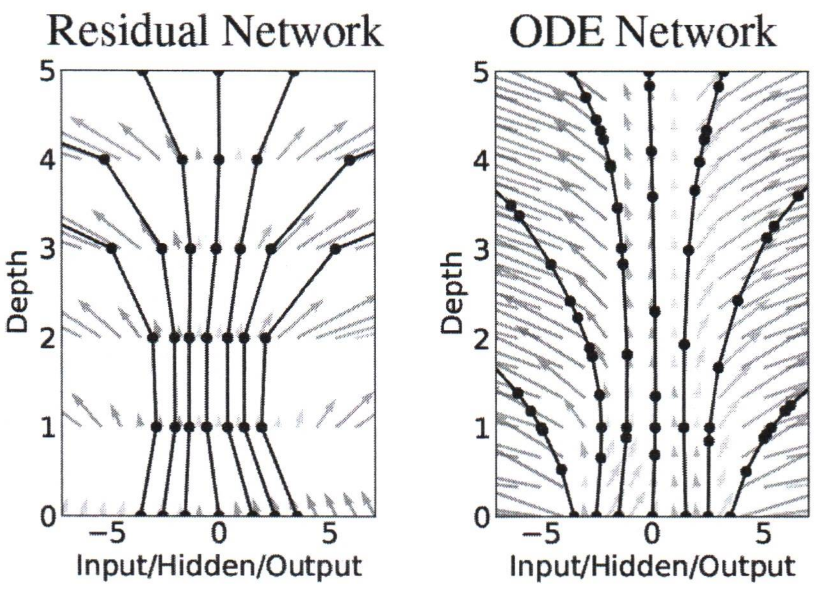
And we can describe its dynamics via

$$\frac{da(t)}{dt} = -a(t)^\intercal \frac{\partial G(\mathbf{h}(t), t, \theta)}{\partial \mathbf{h}}$$

We can compute the derivative of G with respect to $\mathbf{h}$ already - we compute this gradient during backpropagation of traditional neural networks. With this, we can then compute $a(t)$ by using another call to an ODE solver. There is one other derivative, $dL/d\theta$, that can be computed similarly. The paper shows that we can wrap up all of these ODE solves into a single call to an ODE solver, which computes all the necessary gradients for training the system.

What's the point?

Why do we want to do this? According to the paper, we're able to train a model with much less memory, with fewer parameters, and we are able to backpropagate more efficiently. All of these seem like good things! Modern ODE solvers are also adaptive, and can do more work only when needed to get an accurate solution. In the paper they show an experiment where the number of function evaluations that the ODE solver does increases with the number of training epochs - effectively, the system can quickly reach a rough solution, then take more time to refine the training.



An example from the paper showing how using an ode solver can adaptively evaluate the function. Circles represent function evaluations.

I think the most interesting aspect is that treating our system like a continuous time model, allows us to predict continuous time systems. They show a way to take data which arrives at arbitrary times, rather than at fixed intervals, and they can predict the output at arbitrary future times. They test this on fairly simple synthetic data, predicting trajectories of spirals, but get really nice results. I definitely want to see more of this type of work in the future, on larger real-world problems, to see how it does. Being able to draw on the >100 years of research in solving differential equations could be very useful for a young field like deep learning, and hey, I just find the math neat.